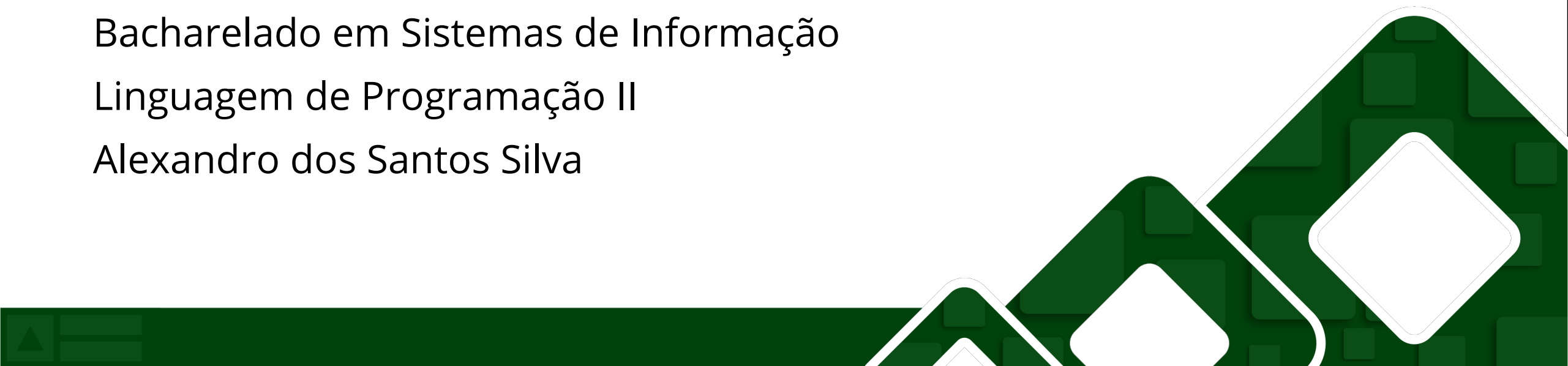


ARQUIVOS E FLUXOS

Bacharelado em Sistemas de Informação

Linguagem de Programação II

Alexandro dos Santos Silva



SUMÁRIO

- Fluxos
- Combinação de Fluxos
- Fluxos de Caracteres
 - Armazenamento de Objetos em Registros de Texto
 - Codificação de Caracteres
- Fluxos de Dados Binários
 - Armazenamento de Objetos em Formato Binário
 - Serialização de Objetos

INTRODUÇÃO

- Caráter temporário de dados armazenados em variáveis e *arrays*
- Armazenamento permanente de dados através de dispositivos de memória secundária (discos, por exemplo)
 - Arquivos: abstração adotada para a gestão de espaço de armazenamento, sendo compreendidos, para tal, como unidades lógicas de informação

FLUXOS

- Abstração para produção (escrita) ou consumo (leitura) de informações
 - **Fluxo de entrada:** objeto a partir do qual pode ser lida uma sequência de bytes
 - **Fluxo de saída:** objeto no qual pode ser escrita uma sequência de bytes
- Origens e destinos de sequências de bytes normalmente associados à arquivos
 - Possibilidade, no entanto, de vinculação com conexões de rede, blocos de memória ou mesmo dispositivos físicos
- Fluxos predefinidos no ambiente de tempo de execução e encapsulados pela classe `java.lang.System`

Fluxo	Descrição
<code>System.in</code>	Fluxo de entrada padrão associado normalmente com operações de teclado
<code>System.out</code>	Fluxo de saída padrão (direcionado normalmente para uma interface de console ou tela)
<code>System.err</code>	Fluxo de erro padrão (direcionado normalmente para uma interface de console ou tela)

FLUXOS

- Hierarquia de classes de entrada e saída de fluxos encabeçadas por duas classes abstratas: `java.io.InputStream` e `java.io.OutputStream`
- Métodos fundamentais

Classe	Método	Descrição
<code>InputStream</code>	<code>abstract int read()</code>	Leitura de 1 (um) byte, seguindo-se a isso retorno deste byte como inteiro ou -1 se for constatado que já tenha sido alcançado final da origem do fluxo de entrada
<code>OutputStream</code>	<code>abstract void write(int b)</code>	Escrita de 1 (um) byte em local de saída

- Implicação no bloqueio do *thread* corrente ao invocar métodos **read** e **write** (até que byte seja lido ou escrito)
 - Oportunidade de outras *threads* fazerem algum processamento enquanto *thread* bloqueado aguarda pela conclusão da operação de leitura ou escrita

FLUXOS

- Implementação dos métodos abstratos **read** e **write** por classes concretas que herdam direta ou indiretamente as classes abstratas **InputStream** e **OutputStream**
 - Leitura, por exemplo, de byte de um arquivo usando-se a classe **FileInputStream**
 - Possibilidade de lançamento de exceções da classe **IOException**
- Fechamento de fluxo após conclusão de operações de leitura ou escrita através do método **close**
 - Liberação de recursos alocados pelo sistema operacional para a realização das operações
 - Em se tratando de fluxo de saída, liberação adicional de *buffer* alocado para o armazenamento temporário de bytes a serem escritos mas que por alguma razão não foram até então efetivamente entregues para escrita (caso ainda hajam bytes nesta condição, eles serão enviados para escrita antes da liberação definitiva do *buffer*)

FLUXOS

- Algumas das subclasses de **InputStream** e **OutputStream** que implementam diferentes fluxos de entrada ou saída de bytes

Classe	Descrição
BufferedInputStream	Fluxo de entrada armazenado em <i>buffer</i>
BufferedOutputStream	Fluxo de saída armazenado em <i>buffer</i>
ByteArrayInputStream	Fluxo de entrada que lê de um <i>array</i> de bytes
ByteArrayOutputStream	Fluxo de saída que grava em um <i>array</i> de bytes
DataInputStream	Fluxo de entrada que dispõe de métodos para a leitura de tipos de dados primitivos
DataOutputStream	Fluxo de saída que dispõe de métodos para a gravação de tipos de dados primitivos
FileInputStream	Fluxo de entrada que lê em um arquivo
FileOutputStream	Fluxo de saída que grava em um arquivo
ObjectInputStream	Fluxo de entrada para objetos
ObjectOutputStream	Fluxo de saída para objetos
PrintStream	Fluxo de saída que dispõe de métodos print e println (objeto predefinido System.in é uma instância desta classe)
PushbackInputStream	Fluxo de entrada que permite que bytes sejam retornados para o fluxo
SequenceInputStream	Fluxo de entrada a partir da combinação de dois ou mais fluxos de entrada que serão lidos sequencialmente (um após o outro)

FLUXOS

- **FileInputStream**: uma das subclasses de **InputStream** com a qual bytes podem ser lidos de arquivos

- Construtor com indicação de nome ou caminho completo de arquivo a ser lido

```
FileInputStream fluxo = new FileInputStream("arquivo.bin");
```

- Interpretação, pelas classes do pacote **java.io**, de caminhos relativos de arquivos a partir de diretório de trabalho do usuário obtido pela chamada de **System.getProperty("user.dir")** (em caso de não especificação de caminhos absolutos)
- Recomendação de invocação da constante estática **java.io.File.separator** para garantir a portabilidade de programas quando caminhos relativos ou absolutos de arquivos incluir um ou mais diretórios (além de que, em se tratando de caminhos de arquivos na plataforma Windows, caractere "\" também representar um caractere de escape)
- Exemplo de instanciação de fluxo de arquivo localizado em subdiretório "arquivos"

```
FileInputStream fluxo = new FileInputStream("arquivos" + File.separator + "arquivo.bin");
```

- Possibilidade de lançamento de exceção da classe **FileNotFoundException** (uma subclasse de **IOException**)

FLUXOS

- **Exemplo 01:** leitura de arquivo, byte a byte, através de instância da classe concreta **FileInputStream** (supressão intencional de parte da codificação neste e nos demais exemplos, para fins de simplificação de listagem)

```
01  import java.io.FileInputStream;
02  import java.io.IOException;
03
04  public class LeituraBytes {
05
06      public static void main(String[] args) {
07          try {
08              FileInputStream fluxo = new FileInputStream("arquivo.bin");
09              byte byteLido;
10
11              // leitura de bytes enquanto não se alcançar final do arquivo
12              do {
13                  byteLido = (byte) fluxo.read();
14                  if (byteLido != -1)                // se byte tiver sido lido...
15                      System.out.print((char)byteLido); // listagem de enésimo byte como caractere
16              } while (byteLido != -1);
17
18              fluxo.close();
19          }
20          catch (IOException e) {
21              e.printStackTrace();
22          }
23      }
24
25  }
```

abertura de fluxo de entrada a partir de invocação de construtor com indicação de nome de arquivo a ser lido

leitura de byte

fechamento de fluxo de entrada

FLUXOS

- Existência de outros métodos, mas não abstratos, nas classes **InputStream** e **OutputStream**
 - Leitura ou escrita de *array* de bytes
 - Obtenção de quantidade de bytes disponíveis para leitura (por questões óbvias, aplicável exclusivamente para a classe **InputStream**)
- Readequação de método **main** do **Exemplo 01**

```
try {  
    FileInputStream fluxo = new FileInputStream("arquivo.bin");  
    int qtdBytes = fluxo.available();  
  
    byte[] bytesLidos = new byte[qtdBytes];  
  
    if (fluxo.read(bytesLidos) != -1) {  
        String texto = new String(bytesLidos);  
        System.out.println(texto);  
    }  
  
    fluxo.close();  
}  
catch (IOException e) {  
    e.printStackTrace();  
}
```

obtenção de quantidade de bytes disponíveis para leitura em arquivo

declaração de *array* com tamanho baseado na quantidade de bytes disponíveis para leitura

leitura de array de bytes invocando-se método **read** e passando-se como parâmetro *array* declarado anteriormente

string a partir de decodificação de bytes lidos

FLUXOS

- Hierarquia adicional de classes abstratas encabeçadas por `java.io.Reader` e `java.io.Writer` para leitura e escrita de fluxos de caracteres
- Métodos fundamentais (semelhantes àquelas das classes `InputStream` e `OutputStream`)

Classe	Método	Descrição
Reader	<code>abstract int read()</code>	Leitura de 1 (um) caractere, seguindo-se a isso retorno deste caractere na forma de unidade de código UTF-16 ou -1 se for constatado que já tenha sido alcançado final da origem do fluxo de entrada
Writer	<code>abstract void write(int c)</code>	Escrita de 1 (um) caractere indicado na forma de unidade de código UTF-16

- Unidade de código UTF-16: um inteiro entre 0 (zero) e 65.535 de acordo com formato de codificação de caracteres conhecido como **Unicode**
 - Codificação de caracteres disponível em <https://www.unicode.org/charts>

FLUXOS

- Algumas das subclasses de **Reader** e **Writer** que implementam diferentes fluxos de entrada ou saída de caracteres

Classe	Descrição
BufferedReader	Fluxo de caractere de entrada armazenado em <i>buffer</i>
BufferedWriter	Fluxo de caractere de saída armazenado em <i>buffer</i>
CharArrayReader	Fluxo de entrada que lê de um <i>array</i> de caracteres
CharArrayWriter	Fluxo de saída que grava em um <i>array</i> de caracteres
FileReader	Fluxo de entrada que lê de um arquivo
FileWriter	Fluxo de saída que grava em um arquivo
InputStreamReader	Fluxo de entrada que converte bytes em caracteres
LineNumberReader	Fluxo de saída que conta linhas
OutputStreamWriter	Fluxo de saída que converte caracteres em bytes
PrintWriter	Fluxo de saída que dispõe de métodos print e println (com fins similares aos dos métodos homônimos de System.in)
PushbackReader	Fluxo de entrada que permite que caracteres sejam retornados para o fluxo
StringReader	Fluxo de entrada que lê de um string
StringWriter	Fluxo de saída que grava em um string

FLUXOS: COMBINAÇÃO

- Associação de fluxos de entrada a um arquivo em disco através da classe **FileInputStream** (conforme mostrado no **Exemplo 01**)

```
FileInputStream fluxo = new FileInputStream("numeros.bin");
```

- Suporte às operações de leitura e escrita *apenas* de bytes

```
byte byteLido = (byte) fluxo.read();
```

- Existência de outros tipos de fluxos de entrada e saída (conforme também visto anteriormente), a exemplo de **DataInputStream**, que permite operações de leitura de valores numéricos

```
DataInputStream fluxoDados = . . .; // supressão intencional de invocação de construtor  
double n = fluxoDados.readInt();
```

- Conclusão: inexistência de métodos para leitura de números inteiros na classe **FileInputStream** assim como de métodos para leitura de dados de um arquivo na classe **DataInputStream**

FLUXOS: COMBINAÇÃO

- Mecanismo de divisão de responsabilidades na hierarquia de fluxos de entrada e saída
 - Fluxos com os quais pode-se ler ou escrever bytes em arquivos e outros locais mais incomuns (a exemplo de **FileInputStream**)
 - Fluxos com os quais pode-se tratar bytes lidos para obter tipos de dados mais úteis ou tais dados serem codificados em bytes para escrita (a exemplo de **DataInputStream**)
- Possibilidade de combinação dos dois tipos de fluxos
 - Exemplo: leitura de números inteiros a partir de arquivo criando-se inicialmente fluxo com uma instância de **FileInputStream**, passando-a em seguida como parâmetro para construtor alternativo de **DataInputStream**

```
FileInputStream fluxoArquivo = new FileInputStream("numeros.bin");  
DataInputStream fluxoDados = new DataInputStream(fluxoArquivo);  
int n = fluxoDados.readInt();
```

FLUXOS: COMBINAÇÃO

- **Exemplo 02 (1/2):** leitura de números a partir de fluxo de bytes associado com arquivo em disco enquanto não se alcançar final de arquivo (situação identificada a partir de lançamento de exceção da classe **EOFException**)

```
01 import java.io.DataInputStream;
02 import java.io.EOFException;
03 import java.io.FileInputStream;
04 import java.io.IOException;
05
06 public class LeituraBinariaNumeros {
07
08     public static void main(String[] args) {
09         try {
10             FileInputStream fluxoArquivo = new FileInputStream("numeros.bin");
11             DataInputStream fluxoDados = new DataInputStream(fluxoArquivo);
12
13             boolean finalArquivo = false;
14
15             while (!finalArquivo) {
16                 try {
17                     int numLido = fluxoDados.readInt();
18                     System.out.println(numLido);
19                 }
20                 catch (EOFException e) {
21                     finalArquivo = true;
22                     System.out.println("Arquivo totalmente lido!");
23                 }
24             }
25         }
26     }
27 }
```

abertura de fluxo de entrada de dados
a partir de fluxo de entrada de arquivo

flag de alcance de final de arquivo

leitura de próximo número (lançamento de
EOFException se final de arquivo for alcançado)

atualização de flag de final de arquivo em
caso de lançamento de **EOFException**

FLUXOS: COMBINAÇÃO

- Exemplo 02 (2/2): continuação

```
25
26     fluxoDados.close();
27     fluxoArquivos.close();
28 }
29 catch (IOException e) {
30     e.printStackTrace();
31 }
32 }
33
34 }
```

fechamento de fluxos

captura de **IOException** por inexistência de arquivo ou ainda por tentativa de fechamento de fluxos

- Fluxos de entrada não *bufferizados* por padrão (leitura e processamento imediato de cada byte requisitando-o ao sistema operacional subjacente)
- Possibilidade, no entanto, de requisição de blocos de bytes (ao invés de cada byte individualmente) e armazenamento em *buffer* criando-se fluxo intermediário com a classe **BufferedInputStream**

```
FileInputStream fluxoArquivo = new FileInputStream("numeros.bin");
BufferedInputStream fluxoBuffer = new BufferedInputStream(fluxoArquivo);
DataInputStream fluxoDados = new DataInputStream(fluxoBuffer);
```

Manutenção de chamada de construtor de **DataInputStream** ao final para continuidade de uso dos métodos de leitura de dados numéricos de arquivo contando-se agora, no entanto, com *buffer*

FLUXOS DE CARACTERES

- Distinção de representação de dados em formato binário e de texto
 - Em termos práticos, armazenamento de textos também na forma de bytes de acordo com determinado formato de codificação de caracteres
 - Exemplo ilustrativo: representação de inteiro 1234 e texto "1234" (em notação hexadecimal)

Formato Binário	Formato de Texto (UTF-8)
00 00 04 D2	31 32 33 34

- Conveniência de formato de texto em algumas circunstâncias por formato binário não ser facilmente legível por humanos
- Escolha de alguma codificação de caracteres em caso de adoção do formato de texto
 - Padrão de codificação adotado internamente pela linguagem Java: UTF-16 (conforme citado anteriormente)
 - Possibilidade de escolha de algum formato de codificação distinto no momento de instanciação de objetos de algumas das subclasses de **Reader** e **Writer**

FLUXOS DE CARACTERES

- Classe abstrata base para escrita de caracteres: **Writer** (conforme citado anteriormente)
- Métodos adicionais não abstratos além de **write(int c)** com os quais *arrays* de caracteres ou strings podem ser enviados para escrita (todos também suscetíveis ao lançamento de exceções da classe **IOException**)

Método	Descrição
<code>void write(char[] ac)</code>	Escrita de <i>array</i> de caracteres indicado como parâmetro
<code>void write(String s)</code>	Escrita de string indicada como parâmetro
<code>void write(String s, int i, int qtd)</code>	Escrita de trecho de string indicada como parâmetro (trecho este definido por índice de primeiro caractere a ser escrito e quantidade de caracteres, conforme parâmetros i e qtd)

Invocação, em termos práticos, de método abstrato **write(int c)** sem que seja necessário, portanto, que subclasses de **Writer** tenham que sobrescrever tais métodos

FLUXOS DE CARACTERES

- **FileWriter**: subclasse de **Writer** para escrita de caracteres em arquivo adotando-se formato de codificação padrão

- Construtor com indicação de arquivo no qual ocorrerão operações de escrita (havendo possibilidade de lançamento de exceções da classe **IOException** em caso de falha de abertura de fluxo)

```
FileWriter fluxo = new FileWriter("funcionario.txt");
```

- Construtor alternativo com parâmetro booleano adicional para determinar se caracteres enviados para escrita serão acrescentados ao final do arquivo em vez de sobreporem aqueles até então armazenados naquele arquivo

```
FileWriter fluxo = new FileWriter("funcionario.txt", true);
```

- Exemplo de invocação de métodos de escrita

```
String nome = "Alexandro";  
double salario = 7500;  
  
fluxoArquivo.write(nome + ' ' + salario);
```

caracteres enviados
para escrita



```
Alexandro 7500.0
```

Em termos práticos, conversão de caracteres para bytes de acordo com formato de codificação padrão e, após isso, gravação em arquivo

FLUXOS DE CARACTERES

- Exemplo 03 (1/2): registro de dados de funcionário em arquivo

```
01 import java.io.FileWriter;
02 import java.io.IOException;
03 import java.util.Scanner;
04
05 public class RegistroFuncionario {
06
07     public static void main(String[] args) {
08         Scanner scanner = new Scanner(System.in);
09
10         // entrada de nome e salário de funcionário
11         System.out.println("Dados de Funcionário");
12         System.out.print("Nome...: ");
13         String nome = scanner.nextLine();
14         System.out.print("Salário: ");
15         double salario = scanner.nextDouble();
16
17         try {
18             FileWriter fluxoArquivo = new FileWriter("funcionario.txt", true);
19
20             fluxoArquivo.write(nome + ' ' + salario + System.lineSeparator());
21
22             fluxoArquivo.close();
```

abertura de fluxo de escrita de
caracteres em arquivo sem sobrepor
conteúdo anterior deste mesmo arquivo

escrita de caracteres (precedido da
conversão, quando for o caso, de
valores que não são caracteres)

fechamento de fluxo de escrita

FLUXOS DE CARACTERES

- **Exemplo 03 (2/2):** continuação

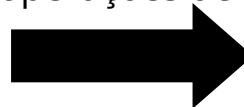
```
23
24         System.out.println("Gravação de dados em arquivo concluída!");
25     }
26     catch (IOException e) {
27         e.printStackTrace();
28     }
29
30     scanner.close();
31 }
32
33 }
```

Uso de método estático **System.lineSeparator** para obtenção de sequência de caracteres de fim de linha apropriado de acordo com sistema operacional)

- Resultado obtido com a execução do programa do **Exemplo 03** (durante a qual são informados dados hipotéticos de funcionário)

```
Dados de Funcionário
Nome...: Alexandro
Salário: 7500,0
Gravação de dados em arquivo concluída!
```

conteúdo de arquivo
após operações de escrita



```
funcionario.txt
Alexandro 7500.0
```

FLUXOS DE CARACTERES

- **PrintWriter**: classe mais conveniente para operações de escrita em fluxos de caracteres
 - Construtor com indicação de arquivo no qual ocorrerão operações de gravação (lançamento de exceção da classe **FileNotFoundException** em caso de arquivo inexistente)

```
PrintWriter fluxo = new PrintWriter("funcionario.txt");
```

- Construtor alternativo com indicação, além de arquivo, de formato de codificação de caracteres

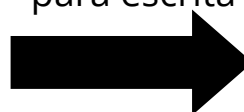
```
PrintWriter fluxo = new PrintWriter("funcionario.txt", "UTF-16");
```

- Gravação de números, caracteres, valores booleanos, strings e objetos em formato de texto utilizando-se dos mesmos métodos **print**, **println** e **printf** disponíveis em **System.out** (sem que haja lançamento de exceções)

```
String nome = "Alexandro";  
double salario = 7500;
```

```
fluxo.print(nome);  
fluxo.print(' ');  
fluxo.print(salario);
```

caracteres enviados
para escrita



```
Alexandro 7500.0
```

Em termos práticos, conversão também de caracteres para bytes de acordo com formato de codificação adotado e, após isso, gravação em arquivo

FLUXOS DE CARACTERES

- Exemplo 04 (1/2): readequação de exemplo anterior com substituição de `FileWriter` por `PrintWriter`

```
01 import java.io.IOException;
02 import java.io.PrintWriter;
03 import java.util.Scanner;
04
05 public class RegistroFuncionario {
06
07     public static void main(String[] args) {
08         Scanner scanner = new Scanner(System.in);
09
10         // entrada de nome e salário de funcionário
11         System.out.println("Dados de Funcionário");
12         System.out.print("Nome...: ");
13         String nome = scanner.nextLine();
14         System.out.print("Salário: ");
15         double salario = scanner.nextDouble();
16
17         try {
18             PrintWriter fluxo = new PrintWriter("funcionario.txt");
19
20             fluxo.print(nome);
21             fluxo.print(' ');
22             fluxo.println(salario);
23
24             fluxo.close();
25             System.out.println("Gravação em arquivo concluída!");
26         }
```

abertura de fluxo de entrada a partir de console

abertura de fluxo de escrita de caracteres em arquivo

escrita de caracteres em fluxo usando métodos `print` e `println` (este último com inclusão, ao final da escrita, de caractere de fim de linha)

fechamento de fluxo de escrita

FLUXOS DE CARACTERES

- **Exemplo 04 (2/2):** continuação

```
27     catch (IOException e) {  
28         e.printStackTrace();  
29     }  
30  
31     scanner.close();  
32 }  
33 }
```

captura de **IOException** por inexistência de arquivo ou ainda por tentativa de fechamento de fluxos de saída

fechamento de fluxo de entrada de console

- Considerações sobre o **Exemplo 04**

- Invocação de método **println** (vide instrução da linha 20): acréscimo de sequência de caracteres de fim de linha apropriado de acordo com sistema operacional
- Descarte de todos os caracteres armazenados anteriormente em arquivo ao abrir novo fluxo
 - Solução: criação de fluxo de saída por **PrintWriter** usando-se construtor alternativo que exige, como parâmetro, fluxo instanciado previamente fornecido por **FileWriter**

```
FileWriter fluxoArquivo = new FileWriter("funcionario.txt", true);  
PrintWriter fluxo = new PrintWriter(fluxoArquivo);
```

- Fluxos de escrita fornecidos por **PrintWriter** associados à *buffer* cuja liberação de caracteres armazenados temporariamente para escrita pode ocorrer de forma automática sempre que **println** for chamado, mas desde que adotado construtor que prevê segundo parâmetro, do tipo booleano (**true**)

```
FileWriter fluxoArquivo = new FileWriter("funcionario.txt", true);  
PrintWriter fluxo = new PrintWriter(fluxoArquivo, true);
```


FLUXOS DE CARACTERES

- **Exemplo 05 (1/2):** escrita, em arquivo de nome "numeros.txt", de sequência de números inteiros fornecida através de interface de entrada padrão (um número por linha)

```
01 import java.io.FileWriter;
02 import java.io.IOException;
03 import java.io.PrintWriter;
04 import java.util.Scanner;
05
06 public class EscritaNumeros {
07
08     public static void main(String[] args) {
09         Scanner scanner = new Scanner(System.in);
10
11         try {
12             FileWriter fluxoArquivo = new FileWriter("numeros.txt", true);
13             PrintWriter fluxo = new PrintWriter(fluxoArquivo, true);
14
15             int n;
16
17             do {
18                 System.out.print("Digite um número ou 0 (zero) para encerrar: ");
19                 n = scanner.nextInt();
20                 if (n != 0)
21                     fluxo.println(n);
22             } while (n != 0);
```

abertura de fluxo de entrada a partir de console

abertura de fluxo de saída de caracteres em arquivo

abertura de fluxo de saída de caracteres com base em fluxo de escrita em arquivo

entrada de número

escrita em fluxo de saída, na forma de caractere, de número se este diferente de 0 (zero) e, após isso, de caracteres de fim de linha

nova entrada de número seguindo-se a isso sua gravação em arquivo enquanto último número for diferente de 0 (zero)

FLUXOS DE CARACTERES

- Exemplo 05 (2/2): continuação

```
23
24     System.out.println("Números digitados gravados em arquivo!");
25
26     fluxo.close();
27     fluxoArquivo.close();
28 }
29 catch (IOException e) {
30     e.printStackTrace();
31 }
32
33 scanner.close();
34 }
35
36 }
```

fechamento de fluxos de saída

captura de **IOException** por inexistência de arquivo ou ainda por tentativa de fechamento de fluxos de saída

fechamento de fluxo de entrada de console

- Resultado obtido com a execução do programa do **Exemplo 06** (durante a qual são informados, sucessivamente, os números 7, 2, 9, 1 e 0)

```
Digite um número ou 0 (zero) para encerrar: 7
Digite um número ou 0 (zero) para encerrar: 2
Digite um número ou 0 (zero) para encerrar: 9
Digite um número ou 0 (zero) para encerrar: 1
Digite um número ou 0 (zero) para encerrar: 0
Números digitados gravados em arquivo!
```

conteúdo de arquivo
após operações de escrita



numeros.txt

```
7
2
9
1
```

FLUXOS DE CARACTERES

- Classe abstrata base para leitura de caracteres: **Reader** (conforme citado anteriormente)
- Métodos adicionais além de **read(int c)** com os quais vários caracteres podem ser lidos (todos também suscetíveis ao lançamento de exceções da classe **IOException**)

Método	Descrição
<code>int read(char[] ac)</code>	Leitura de caracteres armazenando-os em <i>array</i> de caracteres indicado como parâmetro e retornando-se o número de caracteres lidos ou -1 se o final do fluxo tiver sido alcançado
<code>int read(char[] ac, int i, int qtd)</code>	Leitura de caracteres considerando quantidade máxima definida por qtd , armazenando-os em <i>array</i> de caracteres indicado como parâmetro e partir de índice definido por i e retornando-se o número de caracteres lidos ou -1 se o final do fluxo tiver sido alcançado

- Bloqueio de *thread* através do qual métodos de leitura são invocados até que alguma das seguintes condições ocorra
 - Disponibilidade de entrada (caracteres possíveis de serem lidos)
 - Ocorrência de erro de entrada/saída
 - Alcance de final de fluxo de caracteres

FLUXOS DE CARACTERES

- Forma mais simples encontrada para processamento arbitrário de textos através da classe **Scanner**
 - Instanciação de objetos de **Scanner** a partir de qualquer fluxo de entrada (inclusive aqueles associados à arquivos)
 - Instância de **Scanner** a partir de fluxo de entrada de arquivo fornecido por **FileReader**, uma das subclasses de **Reader**

```
Scanner fluxo = new Scanner(new FileReader("numeros.txt"));
```

- **Exemplo 06 (1/2):** leitura e listagem de números armazenados em arquivo linha por linha

```
01 import java.io.FileReader;
02 import java.io.IOException;
03 import java.util.Scanner;
04
05 public class LeituraNumeros {
06
07     public static void main(String[] args) {
08         try {
09             FileReader fluxoArquivo = new FileReader("numeros.txt");
10             Scanner fluxo = new Scanner(fluxoArquivo);
11
12             while (fluxo.hasNext()) {
13                 String linha = fluxo.nextLine();
14                 int n = Integer.parseInt(linha);
15                 System.out.println(n);
16             }
17         }
18     }
19 }
```

abertura de fluxo de leitura de caracteres em arquivo

abertura de fluxo de entrada
com base em fluxo de
leitura de caracteres em arquivo

verificação de existência, ainda, de caracteres não lidos


leitura de próxima linha de caracteres

conversão de linha de caracteres para número inteiro

FLUXOS DE CARACTERES

- **Exemplo 06 (2/2):** continuação

```
17
18     fluxo.close();
19     fluxoArquivo.close();
20 }
21 catch (IOException e) {
22     e.printStackTrace();
23 }
24 }
25
26 }
```



fechamento de fluxos de entrada

- Entre as soluções alternativas para leitura de arquivos de texto, invocação de métodos estáticos disponibilizados por `java.nio.file.Files`

Método	Descrição
<code>String readString(java.nio.file.Path caminho)</code>	Leitura de todos os caracteres de arquivo indicado por caminho , sendo retornados na forma de uma string
<code>List<String> readAllLines(java.nio.file.Path caminho)</code>	Leitura de todas as linhas de arquivo indicado por caminho e retorno das mesmas na forma de uma lista de strings

FLUXOS DE CARACTERES

- Necessidade de passagem de parâmetros do tipo `java.nio.file.Path` ao invocar métodos de `java.nio.file.Files` citados anteriormente
 - Interface usada para representar um caminho hierárquico composto por uma sequência de diretórios e/ou nome de arquivo separados por um separador especial ou delimitador
 - Obtenção de objetos de classes que implementam a interface através da chamada do método estático `java.nio.file.FileSystems.getDefault().getPath`
- **Exemplo 07 (1/2):** readequação de exemplo anterior usando-se a classe `java.nio.file.Files`

```
01 import java.io.IOException;
02 import java.nio.file.FileSystems;
03 import java.nio.file.Files;
04 import java.nio.file.Path;
05 import java.util.Iterator;
06 import java.util.List;
07
08 public class LeituraNumeros {
09
```

FLUXOS DE CARACTERES

- Exemplo 07 (2/2): continuação

```
10 public static void main(String[] args) {
11     try {
12         Path caminho = FileSystems.getDefault().getPath("numeros.txt");
13         List<String> linhas = Files.readAllLines(caminho);
14         Iterator<String> it = linhas.iterator();
15
16         while (it.hasNext()) {
17             String linha = it.next();
18             int n = Integer.parseInt(linha);
19             System.out.println(n);
20         }
21     }
22     catch (IOException e) {
23         e.printStackTrace();
24     }
25 }
26
27
28 }
```

instanciação de caminho de arquivo a ser lido

obtenção, na forma de lista, de todas as linhas de caracteres em arquivo indicado por caminho

iterador de lista de linhas

leitura de próxima linha de caracteres

conversão de linha de caracteres para número inteiro

verificação de existência ainda de linhas não visitadas

FLUXOS DE CARACTERES E OBJETOS

- Adoção de estratégias diversas para armazenamento de instâncias de objetos em arquivos de textos, a exemplo da gravação de valores dos campos de cada instância em uma linha separada e utilizando-se de algum caractere delimitador para separar tais valores
- **Listagem da Classe Funcionario (1/2)**, para fins demonstrativos

```
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;

public class Funcionario {

    private String nome;
    private double salario;
    private GregorianCalendar dataAdmissao;

    public Funcionario(String n, double s, int anoAdmissao, int mesAdmissao, int diaAdmissao) {
        nome = n;
        salario = s;
        dataAdmissao = new GregorianCalendar(anoAdmissao, mesAdmissao - 1, diaAdmissao);
    }

    public String getNome() {
        return nome;
    }
}
```


FLUXOS DE CARACTERES E OBJETOS

- Listagem da Classe Funcionario (2/2): continuação

```
public double getSalario() {
    return salario;
}

public GregorianCalendar getDataAdmissao() {
    return dataAdmissao;
}

public void reajustarSalario(double percentual) {    // reajuste de salário com base em percentual
    if (percentual > 0) {                            // se percentual de reajuste válido...
        double reajuste = salario * percentual / 100; // cálculo de reajuste
        salario += reajuste;                          // incorporação de reajuste ao salário (atualização)
    }
}

public String toString() {
    return "[Nome: " + nome + ", " +
        " Salário: " + salario + ", " +
        " Admissão: " + new SimpleDateFormat("dd/MM/yyyy").format(dataAdmissao.getTime()) + "]\n";
}
}
```

FLUXOS DE CARACTERES E OBJETOS

- Instanciação de objetos de **Funcionario** armazenando-os em lista criada previamente a partir de instância de `java.util.ArrayList`

```
List<Funcionario> quadroFunc = new ArrayList<Funcionario>();  
  
quadroFunc.add(new Funcionario("José Silva", 7500, 1987, 12, 15));  
quadroFunc.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));  
quadroFunc.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));
```

- Registro de objetos em arquivo de texto usando-se uma linha para cada instância e caractere “|” para delimitar valores dos campos de instância

```
José Silva|7500.0|1987-12-15  
Henrique Santos|5000.0|1989-10-1  
Maria Guimarães|7500.0|1990-3-15
```

Emprego de delimitador adicional para separar componentes da data de admissão (ano, mês e dia) de cada funcionário

FLUXOS DE CARACTERES E OBJETOS

- Escrita de objeto em arquivo de texto usando-se métodos da classe `PrintWriter`

```
Funcionario func = . . . ;
int diaAdmissao = func.getDataAdmissao().get(Calendar.DAY_OF_MONTH) ;
int mesAdmissao = func.getDataAdmissao().get(Calendar.MONTH) + 1;
int anoAdmissao = func.getDataAdmissao().get(Calendar.YEAR) ;

fluxo.print(func.getNome() + "|");
fluxo.print(func.getSalario() + "|");
fluxo.println(anoAdmissao + "-" + mesAdmissao + "-" + diaAdmissao);
```

Escrita do valor de cada campo de instância seguindo-se a isso escrita de caractere “|” ou, em sendo o último campo, de caractere de fim de linha

- Escrita de objeto precedida da invocação, além disso, de método `get` da classe `java.util.GregorianCalendar` para extração de componentes da data de admissão
 - Indicação de cada componente da data através de constantes estáticas definidas em `java.util.Calendar`

FLUXOS DE CARACTERES E OBJETOS

- **Exemplo 08 (1/2):** escrita de objetos de **Funcionario** em arquivo de texto

```
01 import java.io.IOException;
02 import java.io.PrintWriter;
03 import java.util.ArrayList;
04 import java.util.Calendar;
05 import java.util.Iterator;
06 import java.util.List;
07
08 public class RegistroQuadroFuncionarios {
09
10     public static void main(String[] args) {
11         try {
12             PrintWriter fluxo = new PrintWriter("quadro-funcionarios.txt");
13
14             List<Funcionario> quadroFunc = new ArrayList<Funcionario>();
15
16             quadroFunc.add(new Funcionario("José Silva", 7500, 1987, 12, 15));
17             quadroFunc.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));
18             quadroFunc.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));
19
20             Iterator<Funcionario> it = quadroFunc.iterator();
```

instanciação de lista

instanciação de objetos
de **Funcionario** e inclusão
em lista

obtenção de iterador da lista

FLUXOS DE CARACTERES E OBJETOS

- Exemplo 08 (2/2): continuação

```
21
22     while (it.hasNext()) {
23         Funcionario func = it.next() ;
24         int diaAdmissao = func.getDataAdmissao().get(Calendar.DAY_OF_MONTH) ;
25         int mesAdmissao = func.getDataAdmissao().get(Calendar.MONTH) + 1 ;
26         int anoAdmissao = func.getDataAdmissao().get(Calendar.YEAR) ;
27
28         {
29             fluxo.print(func.getNome() + "|") ;
30             fluxo.print(func.getSalario() + "|") ;
31             fluxo.println(anoAdmissao + "-" + mesAdmissao + "-" + diaAdmissao) ;
32         }
33         fluxo.close() ;
34     }
35     catch(IOException e) {
36         e.printStackTrace() ;
37     }
38 }
39
40 }
```

verificação de existência de funcionários ainda não visitados

obtenção de próximo funcionário

extração de componentes da data de admissão do funcionário (dia, mês e ano)

escrita de dados de funcionário em fluxo de saída

FLUXOS DE CARACTERES E OBJETOS

- Instanciação de objeto de **Funcionario** armazenado em arquivo de texto lendo-se próxima linha e, após isso, separando sequência de caracteres lidos em subsequências (tokens) com o método **String.split**

```
Scanner fluxo = . . .;

String linha = fluxo.nextLine();

String[] tokens = linha.split("\\|");

String nome      = tokens[0];
double salario = Double.parseDouble(tokens[1]);

String dataAdmissao = tokens[2];
String[] tokensDataAdmissao = dataAdmissao.split("-");
int anoAdmissao = Integer.parseInt(tokensDataAdmissao[0]);
int mesAdmissao = Integer.parseInt(tokensDataAdmissao[1]);
int diaAdmissao = Integer.parseInt(tokensDataAdmissao[2]);

Funcionario func = new Funcionario(nome, salario, anoAdmissao, mesAdmissao, diaAdmissao);
```

Retorno, pelo método **String.split**, de *array* de strings com os tokens obtidos a partir de separação dos caracteres considerando expressão regular de delimitação indicada na forma de parâmetro

FLUXOS DE CARACTERES E OBJETOS

- **Exemplo 09 (1/2):** leitura, instanciação e listagem de objetos de **Funcionario** armazenados em arquivo de texto

```
01 import java.io.FileReader;
02 import java.io.IOException;
03 import java.util.ArrayList;
04 import java.util.List;
05 import java.util.Scanner;
06
07 public class LeituraQuadroFuncionarios {
08
09     public static void main(String[] args) {
10         List<Funcionario> quadroFunc = new ArrayList<Funcionario>();
11
12         try {
13             FileReader fluxoArquivo = new FileReader("quadro-funcionarios.txt");
14
15             Scanner fluxo = new Scanner(fluxoArquivo);
16
17             while (fluxo.hasNext()) {
18                 String linha = fluxo.nextLine();
19
20                 String[] tokens = linha.split("\\|");
21
22                 String nome = tokens[0];
23                 double salario = Double.parseDouble(tokens[1]);
24             }
19         }
20     }
21 }
22
23
24
```

abertura de fluxo de entrada com base em fluxo de leitura de caracteres em arquivo

verificação de existência, ainda, de caracteres não lidos

leitura de próxima linha de caracteres

extração de partes (tokens) da linha delimitados pelo caractere "|"

nome do funcionário (primeiro token)

salário do funcionário (segundo token)

FLUXOS DE CARACTERES E OBJETOS

- Exemplo 09 (2/2): continuação

```
25      String dataAdmissao = tokens[2];
26      String[] tokensDataAdmissao = dataAdmissao.split("-");
27      int anoAdmissao = Integer.parseInt(tokensDataAdmissao[0]);
28      int mesAdmissao = Integer.parseInt(tokensDataAdmissao[1]);
29      int diaAdmissao = Integer.parseInt(tokensDataAdmissao[2]);
30
31      Funcionario func = new Funcionario(nome, salario, anoAdmissao, mesAdmissao, diaAdmissao);
32
33      quadroFunc.add(func);
34  }
35
36  fluxo.close();
37
38  Iterator<Funcionario> it = quadroFunc.iterator();
39
40  while (it.hasNext())
41      System.out.println(it.next());
42  }
43  catch(IOException e) {
44      e.printStackTrace();
45  }
46  }
47
48  }
```

data de admissão do funcionário na forma de string (terceiro token)

extração de partes (tokens) da data de admissão delimitados pelo caractere "-"

instanciação de objeto **Funcionario** com os dados extraídos da linha lida

inclusão de objeto na lista de funcionários

obtenção de iterador da lista de funcionários

obtenção de valores numéricos dos componentes da data de admissão (ano, mês e dia)

enquanto houver funcionários a iterar

obtenção e listagem de próximo funcionário

FLUXOS DE CARACTERES: CODIFICAÇÃO

- Importância da forma como caracteres são codificados em bytes (ao final, caracteres são armazenados como bytes apesar de tratados e processados como tal)
- Adoção, pela linguagem Java, do padrão **Unicode** para a codificação de caracteres
 - Abrangência de caracteres de todos os sistemas de escrita até então conhecidos
 - Variante mais comum da codificação conhecida como **UTF-8** (cada caractere associado com um número de 21 bits)
 - **UTF-16**: outra variante da codificação com a qual usam-se 1 ou 2 valores de 16 bits para codificar cada caractere, sendo esta usada em strings da linguagem Java
- Algumas outras codificações que cobrem apenas determinados subconjuntos de caracteres
 - **ASCII**: codificação tradicional de caracteres do idioma inglês que adota 7 bits (compatível com o UTF-8)
 - **ISO 8859-1**: codificação de 1 byte que inclui caracteres acentuados usados em idiomas da Europa Ocidental
 - **Shift-JIS**: codificação de largura variável para caracteres japoneses

FLUXOS DE CARACTERES: CODIFICAÇÃO

- Indicação de codificações de caracteres suportadas pela linguagem Java através de variáveis estáticas do tipo `java.nio.charset.Charset` declaradas em `java.nio.charset.StandardCharsets`

Variável	Codificação de Caracteres
<code>StandardCharsets.UTF_8</code>	UTF-8
<code>StandardCharsets.UTF_16</code>	UTF-16, com ordem de bytes definida por “marca de ordem de byte”
<code>StandardCharsets.UTF_16BE</code>	UTF-16 em formato <i>big-endian</i> (bytes mais significativos à esquerda)
<code>StandardCharsets.UTF_16LE</code>	UTF-16 em formato <i>little-endian</i> (bytes mais significativos à direita)
<code>StandardCharsets.ISO_8859_1</code>	ISO 8859-1
<code>StandardCharsets.US_ASCII</code>	ASCII

- Formatos ***Big-Endian*** e ***Little-Endian*** distintos entre si pela ordem de posicionamento dos bytes de cada caractere codificado; tome-se, por exemplo, valor de 16 bits representado na forma hexadecimal por 0x2122 (0x21 como byte mais significativo e 0x22 como byte menos significativo)
 - Big-Endian***: 0x21 seguido por 0x22 (0x21 0x22)
 - Little-Endian***: 0x22 seguido por 0x21 (0x22 0x21)
- Marca de Ordem de Byte**: valor de 16 bits colocado no início do arquivo ou do fluxo para definir qual dos dois formatos é usado (*Big-Endian* ou *Little-Endian*)

FLUXOS DE CARACTERES: CODIFICAÇÃO

- Obtenção de instância de **Charset** para alguma outra codificação de caracteres através do método estático **Charset.forName**

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

- Parâmetro adicional em construtores alternativos de algumas subclasses de **Reader** e **Writer** para indicar codificação de caracteres adotada

```
FileWriter fluxo = new FileWriter("funcionario.txt", StandardCharsets.UTF_8);
```

LEITURA E ESCRITA DE DADOS BINÁRIOS

- Melhor eficiência no processamento e transmissão de dados em formato binário apesar de não legível para humanos
- Interface `java.io.DataOutput`: métodos de escrita de números, caracteres, booleanos e strings em formato binário

Método	Descrição
<code>void writeBoolean(boolean v)</code>	Escrita de um valor booleano
<code>void writeByte(int v)</code>	Escrita de um byte indicado na forma de inteiro (serão considerados 8 bits menos significativos)
<code>void writeBytes(String s)</code>	Escrita de uma string alocando-se 1 (um) byte para cada caractere
<code>void writeChar(int v)</code>	Escrita de um caractere alocando-se 2 (dois) bytes
<code>void writeChars(String s)</code>	Escrita de uma string alocando-se 2 (dois) bytes para cada caractere
<code>void writeDouble(double v)</code>	Escrita de um número de ponto flutuante de dupla precisão comprimido em 8 (oito) bytes
<code>void writeFloat(float v)</code>	Escrita de um número de ponto flutuante de precisão simples comprimido em 4 (quatro) bytes
<code>void writeInt(int v)</code>	Escrita de um número inteiro comprimido em 4 (quatro) bytes
<code>void writeLong(long v)</code>	Escrita de um número inteiro longo comprimido em 8 (oito) bytes
<code>void writeShort(int v)</code>	Escrita de um número inteiro curto comprimido em 2 (dois) bytes
<code>void writeUTF(String s)</code>	Escrita de uma string alocando-se 2 (dois) bytes para cada caractere e usando-se versão modificada do formato UTF-8

LEITURA E ESCRITA DE DADOS BINÁRIOS

- Interface `java.io.DataInput`: métodos de leitura de números, caracteres, booleanos e strings em formato binário

Método	Descrição
<code>boolean readBoolean()</code>	Leitura de um byte, seguindo-se a isso retorno de true se byte não for 0 (zero) ou false , caso contrário
<code>byte readByte()</code>	Leitura de um byte seguindo de retorno do mesmo na forma de inteiro
<code>char readChar()</code>	Leitura de 2 (dois) bytes seguido de retorno de caractere correspondente
<code>double readDouble()</code>	Leitura de 8 (oito) bytes seguido de retorno de número de ponto flutuante de dupla precisão correspondente
<code>float readFloat()</code>	Leitura de 4 (quatro) bytes seguido de retorno de número de ponto flutuante de precisão simples correspondente
<code>int readInt()</code>	Leitura de 4 (quatro) bytes seguido de retorno de número inteiro correspondente
<code>String readLine()</code>	Leitura de bytes em quantidade suficiente até encontrar fim de linha ou de arquivo, seguindo-se a isso retorno de string contendo tais bytes convertidos em caracteres
<code>long readLong()</code>	Leitura de 8 (oito) bytes seguido de retorno de número inteiro longo correspondente
<code>short readShort()</code>	Leitura de 2 (dois) bytes seguido de retorno de número inteiro curto correspondente
<code>String readUTF()</code>	Leitura de bytes em quantidade suficiente para retornar string baseada em versão modificada do formato UTF-8

LEITURA E ESCRITA DE DADOS BINÁRIOS

- Algumas das classes que implementam as interfaces `java.io.DataInput` e `java.io.DataOutput`:
`java.io.DataInputStream` (vide **Exemplo 02**) e `java.io.DataOutputStream`
- **Exemplo 10 (1/2)**: escrita de números inteiros em arquivo em formato binário

```
01 import java.io.DataOutputStream;
02 import java.io.FileOutputStream;
03 import java.io.IOException;
04 import java.util.Scanner;
05
06 public class EscritaBinariaNumeros {
07
08     public static void main(String[] args) {
09         Scanner scanner = new Scanner(System.in);
10
11         try {
12             FileOutputStream fluxoArquivo = new FileOutputStream("numeros.bin");
13             DataOutputStream fluxoDados = new DataOutputStream(fluxoArquivo);
14
15             int n;
16
17             do {
18                 System.out.print("Digite um número ou 0 (zero) para encerrar: ");
19                 n = scanner.nextInt();
```

abertura de fluxo de entrada a partir de console

abertura de fluxo de saída
de dados em formato binário com
base em fluxo de escrita em arquivo

LEITURA E ESCRITA DE DADOS BINÁRIOS

- Exemplo 10 (2/2): continuação

```
20         if (n != 0)
21             fluxoDados.writeInt(n);
22     } while (n != 0);
23
24     System.out.println("Números digitados gravados em arquivo!");
25
26     fluxoDados.close();
27     fluxoArquivo.close();
28 }
29 catch (IOException e) {
30     e.printStackTrace();
31 }
32
33 scanner.close();
34 }
35
36 }
```

escrita em fluxo de saída de número informado se este diferente de 0 (zero)

fechamento de fluxos de saída de arquivo e de dados

captura de **IOException** por inexistência de arquivo ou ainda por tentativa de fechamento de fluxos de saída de arquivo e de dados

fechamento de fluxo de entrada de console

Utilização de fluxo predefinido por **System.in** para que números escritos em arquivo sejam fornecidos a partir de interface de console

- Encerramento da entrada de números para escrita quando próximo número informado for 0 (zero)

LEITURA E ESCRITA DE DADOS BINÁRIOS

- Leitura ou escrita de dados em qualquer local em um arquivo através da classe `java.io.RandomAccessFile` (em exemplos anteriores, **acesso de forma exclusivamente sequencial**)
 - Construtor com dois parâmetros: um deles para o nome e/ou caminho do arquivo e outro para modo de acesso (apenas para leitura, se for usada a string "r", ou tanto para leitura como para escrita se for usada a string "rw")
- ```
RandomAccessFile fluxoLeitura = new RandomAccessFile("numeros.bin", "r");
RandomAccessFile fluxoLeituraEscrita = new RandomAccessFile("numeros.bin", "rw");
```
- Ponteiro de arquivo para indicar posição de próximo byte a ser lido ou escrito, podendo ser manipulado através dos seguintes métodos

| Método                             | Descrição                                                                                                                                                                            |
|------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>long getFilePointer()</code> | Retorno de posição atual do ponteiro de arquivo                                                                                                                                      |
| <code>void seek(long pos)</code>   | Posição na qual ocorrerá próxima operação de leitura ou escrita, sendo indicada por <b>pos</b> (deslocamento de bytes representado por este parâmetro a partir do início do arquivo) |

- Mais um método útil: `length()`, com o qual é retornada quantidade de bytes do arquivo
- Implementação, pela classe, das interfaces `DataInput` e `DataOutput` (com o que são aplicáveis métodos citados na seção anterior, tais como `readInt` / `writeInt` e `readChar` / `writeChar`)



# LEITURA E ESCRITA DE DADOS BINÁRIOS

- **Exemplo 11 (1/2):** implementação de leitura aleatória de números inteiros armazenados em arquivo em formato binário (4 bytes por inteiro) utilizando-se do programa do exemplo anterior

```
01 import java.io.IOException;
02 import java.io.RandomAccessFile;
03 import java.util.Scanner;
04
05 public class LeituraAleatoriaNumeros {
06
07 public static void main(String[] args) {
08 final int BYTES_POR_NUMERO = 4;
09
10 try {
11 Scanner scanner = new Scanner(System.in);
12 RandomAccessFile fluxoArquivo = new RandomAccessFile("numeros.bin", "r");
13
14 long tamArquivo = fluxoArquivo.length();
15 long qtdNumeros = tamArquivo / BYTES_POR_NUMERO;
16 long pos;
17 }
18 }
19 }
```

bytes por número inteiro armazenado em arquivo

abertura de fluxo de entrada a partir de console

fluxo de leitura de arquivo com acesso aleatório

total de bytes do fluxo de arquivo

quantidade de números inteiros em arquivo considerando total de bytes do arquivo e quantidade de bytes por número

# LEITURA E ESCRITA DE DADOS BINÁRIOS

- Exemplo 11 (2/2): continuação

```
18 do {
19 System.out.print("Digite posição (1-" + qtdNumeros +
20 ") de número a ser lido ou 0 (zero) para encerrar: ");
21 pos = scanner.nextLong();
22 if (pos != 0) {
23 fluxoArquivo.seek((pos - 1) * BYTES_POR_NUMERO);
24 System.out.println("Número lido: " + fluxoArquivo.readInt());
25 }
26 } while (pos != 0);
27
28 fluxoArquivo.close();
29 scanner.close();
30 }
31 catch (IOException e) {
32 e.printStackTrace();
33 }
34 }
35
36 }
```

entrada de posição de número a ser lido em arquivo

mudança de ponteiro de arquivo para leitura de acordo com posição do número a ser lido

leitura de número em fluxo de arquivo

nova leitura em fluxo de arquivo enquanto posição de próximo número a ser lido for diferente de 0 (zero)

# LEITURA E ESCRITA DE OBJETOS EM FORMATO BINÁRIO

- Armazenamento, em formato binário e na forma de registros de mesmo tamanho, dos mesmos objetos de **Funcionario** do **Exemplo 08**
  - **Desafio:** valores inteiros e de ponto flutuante representados em formato binário com quantidade fixa de bytes, mas tal raciocínio não aplicável às strings para as quais tamanho depende da quantidade de caracteres
  - **Solução:** fixação do comprimento da string que armazena o nome do funcionário em 40 caracteres (se ela possuir mais que 40 caracteres, caracteres em excesso seriam removidos; caso contrário, caracteres de espaço seriam acrescentados até que sejam completados 40 caracteres)
- Composição de cada registro da classe **Funcionario** em formato binário

| Campo de Instância | Tipo de Dado                                | Quantidade de Bytes |
|--------------------|---------------------------------------------|---------------------|
| nome               | String de 40 caracteres                     | 80                  |
| salario            | Número de ponto flutuante de dupla precisão | 8                   |
| dataAdmissao       | 3 números inteiros (ano, mês e dia)         | 12                  |
| <b>TOTAL</b>       |                                             | <b>100</b>          |

# LEITURA E ESCRITA DE OBJETOS EM FORMATO BINÁRIO

- Readequação da classe **Funcionario**, pela inclusão de constantes estáticas e de método que retorna nome de funcionário com quantidade de caracteres determinada por uma destas constantes

```
public class Funcionario {

 public static final int TAMANHO_NOME = 40;
 public static final int BYTES_POR_REGISTRO = 100;
 private String nome;

 . . .

 public String getNomeTamFixo() {
 String nomeTemp = nome;

 if (nomeTemp.length() > TAMANHO_NOME)
 nomeTemp = nomeTemp.substring(0, TAMANHO_NOME);
 else
 for (int i = nomeTemp.length(); i < TAMANHO_NOME; i++)
 nomeTemp += " ";

 return nomeTemp;
 }
}
```

constante para indicar quantidade máxima de caracteres do nome

constante para indicar quantidade de bytes para armazenar registro de cada instância da classe

supressão de parte de campos de instância e métodos listados nas págs. 32 e 33

verificação de nome com mais caracteres que tamanho máximo indicado em constante

extração de caracteres limitada ao tamanho máximo

inclusão de caracteres de espaço em branco se nome conter caracteres em quantidade menor do que aquela indicada por constante

# LEITURA E ESCRITA DE OBJETOS EM FORMATO BINÁRIO

- **Exemplo 12 (1/2):** readequação de **Exemplo 08**, pela escrita de registros de **Funcionario** em formato binário

```
01 import java.io.IOException;
02 import java.io.RandomAccessFile;
03 import java.util.ArrayList;
04 import java.util.Calendar;
05 import java.util.Iterator;
06 import java.util.List;
07
08 public class RegistroBinarioQuadroFuncionarios {
09
10 public static void main(String[] args) {
11 try {
12 RandomAccessFile fluxoArquivo = new RandomAccessFile("quadro-funcionarios.dat", "rw");
13
14 List<Funcionario> quadroFunc = new ArrayList<Funcionario>();
15
16 quadroFunc.add(new Funcionario("José Silva", 7500, 1987, 12, 15));
17 quadroFunc.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));
18 quadroFunc.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));
19
20 Iterator<Funcionario> it = quadroFunc.iterator();
21 }
22 }
23 }
```

fluxo de leitura e escrita de arquivo com acesso aleatório

iterador de lista de funcionários

# LEITURA E ESCRITA DE OBJETOS EM FORMATO BINÁRIO

- Exemplo 12 (2/2): continuação

```
22 while (it.hasNext()) {
23 Funcionario func = it.next();
24
25 int diaAdmissao = func.getDataAdmissao().get(Calendar.DAY_OF_MONTH);
26 int mesAdmissao = func.getDataAdmissao().get(Calendar.MONTH) + 1;
27 int anoAdmissao = func.getDataAdmissao().get(Calendar.YEAR);
28
29 fluxoArquivo.writeChars(func.getNomeTamFixo());
30 fluxoArquivo.writeDouble(func.getSalario());
31 fluxoArquivo.writeInt(anoAdmissao);
32 fluxoArquivo.writeInt(mesAdmissao);
33 fluxoArquivo.writeInt(diaAdmissao);
34 }
35
36 fluxoArquivo.close();
37 }
38 catch(IOException e) {
39 e.printStackTrace();
40 }
41 }
42
43 }
```

obtenção de próximo funcionário

escrita de dados de funcionário em formato binário (nome com tamanho fixo, salário, ano, mês e dia de admissão)

iteração da lista até que seja alcançado último funcionário

# LEITURA E ESCRITA DE OBJETOS EM FORMATO BINÁRIO

- **Exemplo 13 (1/2):** listagem de registros de **Funcionario** armazenados em formato binário conforme implementação listada no exemplo anterior

```
01 import java.io.IOException;
02 import java.io.RandomAccessFile;
03 import java.util.ArrayList;
04 import java.util.Iterator;
05 import java.util.List;
06
07 public class LeituraBinariaQuadroFuncionarios {
08
09 public static void main(String[] args) {
10 List<Funcionario> quadroFunc = new ArrayList<Funcionario>();
11
12 try {
13 RandomAccessFile fluxoArquivo = new RandomAccessFile("quadro-funcionarios.dat", "r");
14
15 long tamArquivo = fluxoArquivo.length();
16 long qtdRegistros = tamArquivo / Funcionario.BYTES_POR_REGISTRO;
17
18 for (int i = 0; i < qtdRegistros; i++) {
19 String nome = "";
20
21 for (int j = 0; j < Funcionario.TAMANHO_NOME; j++)
22 nome += fluxoArquivo.readChar();
23
24 }
```

# LEITURA E ESCRITA DE OBJETOS EM FORMATO BINÁRIO

- Exemplo 13 (2/2): continuação

```
24 double salario = fluxoArquivo.readDouble();
25
26 int anoAdmissao = fluxoArquivo.readInt();
27 int mesAdmissao = fluxoArquivo.readInt();
28 int diaAdmissao = fluxoArquivo.readInt();
29
30 Funcionario func = new Funcionario(nome.trim(), salario, anoAdmissao, mesAdmissao, diaAdmissao);
31
32 quadroFunc.add(func);
33 }
34
35 fluxoArquivo.close();
36
37 Iterator<Funcionario> it = quadroFunc.iterator();
38
39 while (it.hasNext())
40 System.out.println(it.next());
41 }
42 catch(IOException e) {
43 e.printStackTrace();
44 }
45 }
46
47 }
```



# SERIALIZAÇÃO DE OBJETOS

- Inadequação do armazenamento de objetos utilizando-se do formato de registro de tamanho fixo quando tais objetos são de tipos diferentes
  - Exemplo: lista de objetos de **Funcionario** em que alguns deles são instâncias de subclasses
- Escrita de qualquer objeto em um fluxo de saída, com possibilidade de recuperação lendo-o mais tarde e sendo tal mecanismo conhecido como **serialização de objetos**
- Classes para leitura e escrita de objetos: `java.io.ObjectInputStream` e `java.io.ObjectOutputStream`
- Métodos fundamentais (passíveis de lançamento de exceções de `IOException` e, no caso especificamente da leitura de objeto, de `ClassNotFoundException`)

| Classe                          | Método                                    | Descrição                                                                                                                                                  |
|---------------------------------|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ObjectInputStream</code>  | <code>Object readObject()</code>          | Leitura de um objeto do fluxo, de modo a incluir assinatura da classe e valores dos campos de instância do respectivo objeto, após o que ele é retornado   |
| <code>ObjectOutputStream</code> | <code>void writeObject(Object obj)</code> | Escrita de um objeto indicado por <code>obj</code> ao fluxo, de modo a incluir assinatura da classe e valores dos campos de instância do respectivo objeto |

# SERIALIZAÇÃO DE OBJETOS

- Listagem da classe **Gerente**, para fins demonstrativos

```
import java.text.SimpleDateFormat;

public class Gerente extends Funcionario {

 private double bonus;

 public Gerente(String n, double s, int anoAdmissao, int mesAdmissao, int diaAdmissao) {
 super(n, s, anoAdmissao, mesAdmissao, diaAdmissao);
 bonus = 0;
 }

 public void setBonus(double bonus) {
 this.bonus = bonus;
 }

 public double getSalario() {
 double salarioBase = super.getSalario();
 return salarioBase + bonus;
 }

 public String toString() {
 return "[Nome: " + super.nome + ", " +
 "Salário: " + (super.salario + bonus) + ", " +
 "Admissão: " + new SimpleDateFormat("dd/MM/yyyy").format(dataAdmissao.getTime()) + "]\n";
 }
}
```

# SERIALIZAÇÃO DE OBJETOS

- Abertura de fluxo de saída através de objeto de `ObjectOutputStream` (associado com fluxo de arquivo)

```
FileOutputStream fluxoArquivo = new FileOutputStream("quadro-funcionarios.obj");
ObjectOutputStream fluxoObjetos = new ObjectOutputStream(fluxoArquivo);
```

- Instanciação de objetos de `Funcionario` e `Gerente` e, após isso, escrita dos mesmos em fluxo

```
Funcionario func = new Funcionario("José Silva", 7500, 1987, 12, 15);
Gerente ger = new Gerente("Renata Batista", 12200, 1990, 3, 15);
ger.setBonus(1000);

fluxoObjetos.writeObject(func);
fluxoObjetos.writeObject(ger);
```

- Abertura de fluxo de entrada através de objeto de `ObjectInputStream` (igualmente vinculado ao mesmo fluxo de arquivo associado ao objeto de `ObjectOutputStream`)

```
FileInputStream fluxoArquivo = new FileInputStream("quadro-funcionarios.obj");
ObjectInputStream fluxoObjetos = new ObjectInputStream(fluxoArquivo);
```

- Recuperação de objetos na mesma ordem em que foram escritos (necessidade de coerção de tipo dado que método `readObject` retorna um objeto de tipo genérico)

```
Funcionario func = (Funcionario) fluxoObjetos.readObject();
Gerente ger = (Gerente) fluxoObjetos.readObject();
```

# SERIALIZAÇÃO DE OBJETOS

- Adicionalmente, necessidade de marcação da classe **Funcionario** como **serializável** através da implementação, por ela, da interface **java.io.Serializable**

```
public class Funcionario implements Serializable {
 . . .
}
```

Não definição, no entanto, de nenhum método pela interface **Serializable** (portanto, inclusão da cláusula **implements** é suficiente para tornar uma classe serializável)

- Lançamento de exceção (**java.io.NotSerializableException**) em caso de tentativa de escrita de objeto não serializado
- Leitura e escrita de valores de tipos primitivos usando-se, por exemplo, métodos **writeInt/readInt** ou **writeDouble/readDouble** (interfaces **DataInput** e **DataOutput** também implementadas, respectivamente, por **ObjectInputStream** e **ObjectOutputStream**)

# SERIALIZAÇÃO DE OBJETOS

- **Exemplo 14 (1/2):** escrita, em formato serializado, de objetos de **Funcionario** e **Gerente** em arquivo, sendo ela precedida por escrita de número inteiro indicativo da quantidade de objetos

```
01 import java.io.FileOutputStream;
02 import java.io.IOException;
03 import java.io.ObjectOutputStream;
04 import java.util.ArrayList;
05 import java.util.Iterator;
06 import java.util.List;
07
08 public class RegistroListaObjetos {
09
10 public static void main(String[] args) {
11 try {
12 FileOutputStream fluxoArquivo = new FileOutputStream("quadro-funcionarios.obj");
13 ObjectOutputStream fluxoObjetos = new ObjectOutputStream(fluxoArquivo);
14
15 List<Funcionario> quadroFunc = new ArrayList<Funcionario>();
16
17 quadroFunc.add(new Funcionario("José Silva", 7500, 1987, 12, 15));
18 quadroFunc.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));
19 quadroFunc.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));
20
21 Gerente gerente = new Gerente("Renata Batista", 12200, 1990, 3, 15);
22 gerente.setBonus(1000);
23 quadroFunc.add(gerente);
```

# SERIALIZAÇÃO DE OBJETOS

- Exemplo 14 (2/2): continuação

```
24
25 fluxoObjetos.writeInt(quadroFunc.size());
26
27 Iterator<Funcionario> it = quadroFunc.iterator();
28
29 while (it.hasNext()) {
30 Funcionario func = it.next();
31
32 fluxoObjetos.writeObject(func);
33 }
34
35 fluxoObjetos.close();
36 fluxoArquivo.close();
37 }
38 catch(IOException e) {
39 e.printStackTrace();
40 }
41 }
42
43 }
```

Ao invocar `writeObject`, consulta de todos os campos de instância dos objetos de **Funcionario** e **Gerente** para que seus conteúdos sejam enviados ao fluxo para serem salvos (nome, salário e data de admissão e, no caso de instâncias de **Gerente**, de bônus)

# SERIALIZAÇÃO DE OBJETOS

- **Exemplo 15 (1/2):** leitura e listagem de objetos de **Funcionario** e **Gerente** armazenados em arquivo em formato serializado, sendo ela precedida por leitura de número inteiro indicativo da quantidade de objetos

```
01 import java.io.FileInputStream;
02 import java.io.IOException;
03 import java.io.ObjectInputStream;
04 import java.util.ArrayList;
05 import java.util.Iterator;
06 import java.util.List;
07
08 public class LeituraListaObjetos {
09
10 public static void main(String[] args) {
11 try {
12 FileInputStream fluxoArquivo = new FileInputStream("quadro-funcionarios.obj");
13 ObjectInputStream fluxoObjetos = new ObjectInputStream(fluxoArquivo);
14
15 List<Funcionario> quadroFunc = new ArrayList<Funcionario>();
16
17 int numFunc = fluxoObjetos.readInt();
18
19 for (int i = 0; i < numFunc; i++) {
20 Funcionario func = (Funcionario)fluxoObjetos.readObject();
21
22 quadroFunc.add(func);
23 }
24 }
25 }
26 }
```

# SERIALIZAÇÃO DE OBJETOS

- **Exemplo 15 (2/2):** continuação

```
24
25 fluxoObjetos.close();
26 fluxoArquivo.close();
27
28 Iterator<Funcionario> it = quadroFunc.iterator();
29
30 while (it.hasNext())
31 System.out.println(it.next());
32 }
33 catch(IOException e) {
34 e.printStackTrace();
35 }
36 catch(ClassNotFoundException e) {
37 e.printStackTrace();
38 }
39 }
40
41 }
```

- Considerações sobre o **Exemplo 15**

- Em coerção de tipo de objeto retornado por `readObject`, possibilidade dele ser de subclasses (como `Gerente`)
- Captura de exceção da `ClassNotFoundException` lançada por `readObject`, por leitura de objeto cuja assinatura de classe não pôde ser encontrada considerando ambiente de execução do programa pela Máquina Virtual Java



# SERIALIZAÇÃO DE OBJETOS

- **Número serial único** associado com cada objeto salvo em um fluxo de saída de objetos
- Número serial de extrema relevância particularmente em se tratando de escrita de objetos que contêm, entre seus campos de instância, referências de outros objetos
  - Readequação da classe **Gerente**, para fins demonstrativos (possibilidade de cada gerente possuir um secretário representado por objeto da superclasse **Funcionario**)

```
public class Gerente extends Funcionario {
 private double bonus;
 private Funcionario secretario;
 . . .
 public Funcionario getSecretario() {
 return secretario;
 }
 public void setSecretario(Funcionario secretario) {
 this.secretario = secretario;
 }
 public String toString() {
 String nomeSecretario = secretario == null ? "nenhum" : secretario.getNome();
 return "[Nome: " + super.nome + ", " +
 "Salário: " + (super.salario + bonus) + ", " +
 "Admissão: " + new SimpleDateFormat("dd/MM/yyyy").format(dataAdmissao.getTime()) + ", " +
 "Secretário: " + nomeSecretario + "]";
 }
}
```

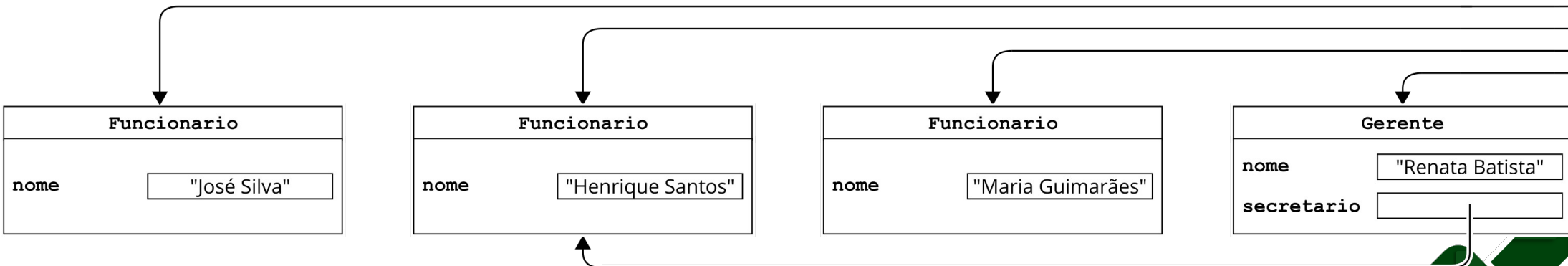
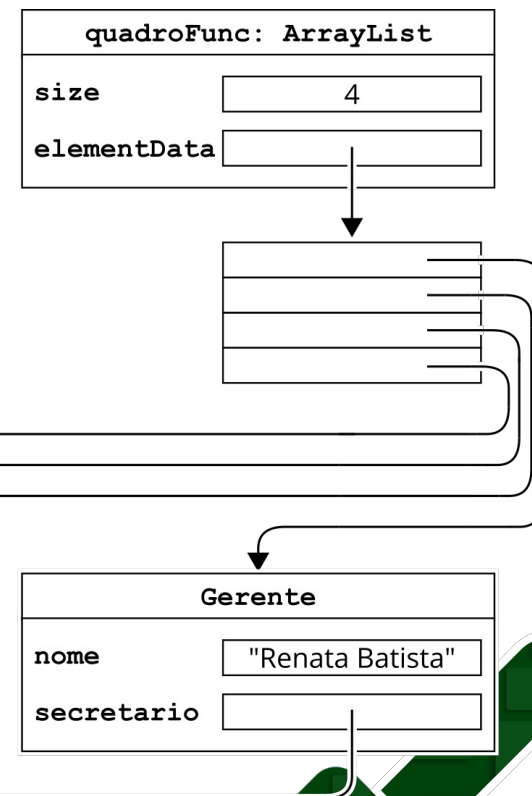
# SERIALIZAÇÃO DE OBJETOS

- Instanciação de 3 (três) objetos que representam funcionários e de 1 (um) gerente cujo secretário é um dos funcionários instanciados anteriormente

```
List<Funcionario> quadroFunc = new ArrayList<Funcionario>();

quadroFunc.add(new Funcionario("José Silva", 7500, 1987, 12, 15));
quadroFunc.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));
quadroFunc.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));

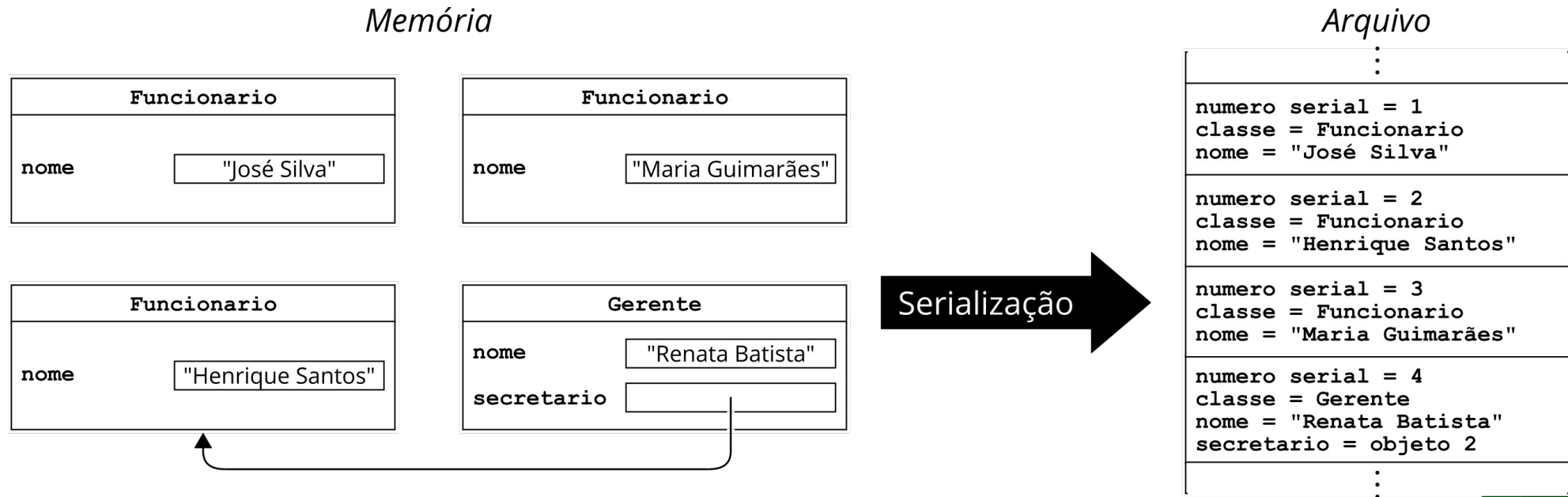
Gerente gerente = new Gerente("Renata Batista", 12200, 1990, 3, 15);
gerente.setBonus(1000);
gerente.setSecretario(quadroFunc.get(1));
quadroFunc.add(gerente);
```



Omissão intencional de parte dos campos de instância dos objetos de **Funcionario** e **Gerente**

# SERIALIZAÇÃO DE OBJETOS

- Mecanismo de serialização de objetos (salvamento em fluxo de objetos)
  1. Associação de um **número serial x** a cada referência de objeto encontrada
  2. Escrita dos dados do objeto no fluxo de saída quando referência de tal objeto for encontrada pela primeira vez
  3. Para cada nova referência daquele objeto que for encontrada, escrita de indicativo ou flag do objeto já ter sido salvo anteriormente com **número serial x**



# SERIALIZAÇÃO DE OBJETOS

- Procedimento reverso ao ler fluxo de entrada de objetos (também conhecido como **deserialização**)
  1. Construção de objeto e inicialização com respectivos dados advindos do fluxo quando ele for especificado pela primeira vez, seguindo-se a isso registro, em memória, da associação da referência daquele objeto com seu **número serial x**
  2. Para cada indicativo ou flag de objeto ter sido salvo anteriormente com o **número serial x**, recuperação de referência de objeto associado àquele número serial
- **Exemplo 16 (1/3):** escrita de objetos de **Funcionario** e **Gerente** em que este último contém referência de outro objeto (**Funcionario**) na condição de secretário

```
01 import java.io.FileOutputStream;
02 import java.io.IOException;
03 import java.io.ObjectOutputStream;
04 import java.util.ArrayList;
05 import java.util.Iterator;
06 import java.util.List;
07
08 public class RegistroListaObjetos {
```

# SERIALIZAÇÃO DE OBJETOS

- Exemplo 16 (2/3): continuação

```
09
10 public static void main(String[] args) {
11 try {
12 FileOutputStream fluxoArquivo = new FileOutputStream("quadro-funcionarios-v2.obj");
13 ObjectOutputStream fluxoObjetos = new ObjectOutputStream(fluxoArquivo);
14
15 List<Funcionario> quadroFunc = new ArrayList<Funcionario>();
16
17 quadroFunc.add(new Funcionario("José Silva", 7500, 1987, 12, 15));
18 quadroFunc.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));
19 quadroFunc.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));
20
21 Gerente gerente = new Gerente("Renata Batista", 12200, 1990, 3, 15);
22 gerente.setBonus(1000);
23 gerente.setSecretario(quadroFunc.get(1));
24 quadroFunc.add(gerente);
25
26 fluxoObjetos.writeInt(quadroFunc.size());
27
28 Iterator<Funcionario> it = quadroFunc.iterator();
29
```

# SERIALIZAÇÃO DE OBJETOS

- Exemplo 16 (3/3): continuação

```
30 while (it.hasNext()) {
31 Funcionario func = it.next();
32
33 fluxoObjetos.writeObject(func);
34 }
35
36 fluxoObjetos.close();
37 fluxoArquivo.close();
38 }
39 catch(IOException e) {
40 e.printStackTrace();
41 }
42 }
43
44 }
```

# SERIALIZAÇÃO DE OBJETOS

- Classes concretas de coleções com suporte à serialização, a exemplo de **ArrayList** (ou seja, igual implementação, por elas, da interface **Serializable**)
- Maior praticidade ao escrever e ler, com isso, todos os elementos da coleção em fluxos de objetos a partir da escrita e leitura apenas da própria instância da coleção
  - Necessidade, no entanto, de todos os elementos da coleção serem serializáveis
  - Escrita e leitura de elementos conforme procedimentos de serialização e deserialização citados anteriormente (tais elementos são referências de objetos)
- **Exemplo 17 (1/2):** escrita de instância de **ArrayList** em fluxo de saída de objetos

```
01 import java.io.FileOutputStream;
02 import java.io.IOException;
03 import java.io.ObjectOutputStream;
04 import java.util.ArrayList;
05 import java.util.List;
06
07 public class RegistroListaObjetos2 {
08
09 public static void main(String[] args) {
10 try {
11 FileOutputStream fluxoArquivo = new FileOutputStream("lista-funcionarios.obj");
```

# SERIALIZAÇÃO DE OBJETOS

- Exemplo 17 (2/2): continuação

```
12 ObjectOutputStream fluxoObjetos = new ObjectOutputStream(fluxoArquivo);
13
14 List<Funcionario> quadroFunc = new ArrayList<Funcionario>();
15
16 quadroFunc.add(new Funcionario("José Silva", 7500, 1987, 12, 15));
17 quadroFunc.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));
18 quadroFunc.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));
19
20 Gerente gerente = new Gerente("Renata Batista", 12200, 1990, 3, 15);
21 gerente.setBonus(1000);
22 quadroFunc.add(gerente);
23
24 fluxoObjetos.writeObject(quadroFunc);
25
26 fluxoObjetos.close();
27 fluxoArquivo.close();
28 }
29 catch(IOException e) {
30 e.printStackTrace();
31 }
32 }
33
34 }
```



# SERIALIZAÇÃO DE OBJETOS

- **Exemplo 18 (1/2):** leitura de instância de **ArrayList** a partir de fluxo de entrada de objetos associado com fluxo de arquivo manipulado pelo programa do **Exemplo 17**

```
01 import java.io.FileInputStream;
02 import java.io.IOException;
03 import java.io.ObjectInputStream;
04 import java.util.ArrayList;
05 import java.util.Iterator;
06 import java.util.List;
07
08 public class LeituraListaObjetos2 {
09
10 public static void main(String[] args) {
11 try {
12 FileInputStream fluxoArquivo = new FileInputStream("lista-funcionarios.obj");
13 ObjectInputStream fluxoObjetos = new ObjectInputStream(fluxoArquivo);
14
15 List<Funcionario> quadroFunc = (ArrayList<Funcionario>) fluxoObjetos.readObject();
16
17 fluxoObjetos.close();
18 fluxoArquivo.close();
19 }
```

# SERIALIZAÇÃO DE OBJETOS

- **Exemplo 18 (2/2):** continuação

```
20 Iterator<Funcionario> it = quadroFunc.iterator();
21
22 while (it.hasNext())
23 System.out.println(it.next());
24 }
25 catch(IOException e) {
26 e.printStackTrace();
27 }
28 catch(ClassNotFoundException e) {
29 e.printStackTrace();
30 }
31 }
33
34 }
```

- Considerações sobre os **Exemplos 17 e 18**

- Ausência de instrução de escrita ou leitura de número inteiro indicativo da quantidade de funcionários e/ou gerentes armazenados em fluxo de arquivo (ao contrário do observado nos **Exemplos 14, 15 e 16**)

# REFERÊNCIAS BIBLIOGRÁFICAS

- DEITEL, Paul; DEITEL, Harvey. Java: **Como Programar**. 10.ed. São Paulo: Pearson Education do Brasil, 2017.
- HORSTMANN, Cay S. **Core Java®**: Volume II - Advanced Features. 10.ed. Upper Saddle River: Prentice Hall, 2017.
- SCHILDT, Herbert. **Programação com Java**: Uma Introdução Abrangente. Porto Alegre: AMGH, 2013.
- TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 4.ed. São Paulo: Pearson Education do Brasil, 2016.